The Four CCA Secure Techniques

Hash-Then-Encrypt (HtE)

This is a technique where the plaintext is hashed first. The hash is then concatenated to the end of the plaintext and lastly, the combined text is encrypted before sending it. Once received, it is decrypted and separated. The hash value is then compared to check if the message has been tampered with.
Hashing itself improves **Integrity**. However, in this technique, if the encryption key were to be leaked, it makes hashing completely useless. The attacker would obtain the plaintext and only need to figure out which hashing algorithm was used to fool the authentication process. So this technique would only have one layer of security.

Mac-Then-Encrypt (MtE)

This technique will generate a MAC of the plaintext using one key. Concatenate the MAC to the plaintext and then encrypt the combined text using another key. Similar to ***Hash-Then-Encrypt,*** once received it will need to be decrypted first and then checked if the message has been tampered with by comparing the MAC. This has the same vulnerability as Hash-Then-Encrypt, but it has better security as it uses a MAC which requires a separate key instead of just a hash. The MAC provides no integrity protection of the ciphertext as it is only after decrypting the message that we can determine its authenticity.

Encrypt-And-Mac (E&M)

This technique encrypts the plaintext using one key and also generates a MAC of the plaintext using another key. These two values are then combined with a comma to separate them in between. This MAC also does not provide integrity protection of the ciphertext as it is calculated based on the plaintext. This can generate a vulnerability to chosen-ciphertext attacks on the cipher such as padding oracles attacks.

With all the above techniques, they have the following efficiencies:

Reduced processing overhead: Since the MAC/Hash is generated over the plaintext data, rather than the encrypted data, the encryption step operates on a smaller amount of data. This can result in reduced processing overhead, especially for encryption algorithms that are computationally demanding.

Flexibility in encryption schemes: As was mentioned before, since these techniques do not provide integrity protection, it is possible to use encryption schemes that do not provide built-in integrity protection, allowing for a wider range of encryption algorithms to be used. This opens up the potential to use more efficient and specialised encryption algorithms.

Encrypt-Then-Mac (EtM)

This technique will encrypt the plaintext first with one key. After that, the ciphertext will be used to generate a MAC with another key. The MAC will then be combined with the ciphertext separated by a comma. The advantages of this technique are:

Provides integrity of both Ciphertext and Plaintext: By using a shared secret MAC key, we can determine if a ciphertext is authentic or forged. This is particularly useful in public-key cryptography scenarios, where anyone can send you messages. EtM guarantees that you only read valid messages by filtering out any invalid ciphertext through the MAC verification.

The MAC does not reveal information about the Plaintext: The MAC appears random just like the cipher output, so the structure of the plaintext is not carried over to the MAC.

Error propagation: If an error occurs during decryption, it will be detected during MAC verification. This eliminates the need for further decryption attempts, saving computational resources.

Because of these reasons, it is the most secure technique out of the four. However, it also comes with its downsides:

Increased overhead: EtM introduces additional overhead due to the generation and transmission of the MAC alongside the encrypted data. This has an impact on the network bandwidth and overall performance, especially for large volumes of data.

Increased latency: Although simplistic and easy to implement, as the MAC generation is an extra step, it adds computational time to the overall process, potentially increasing latency. This trade-off may be more pronounced in scenarios where real-time communication or low-latency requirements are critical.

Our Implementation

For our implementation of a secure protocol, we used a combination of encryption and signing. This combination of these two operations satisfies all the main goals of cryptography which are **Integrity**, **Confidentiality**, **Authenticity,** and **Accountability.** Further on, will explain how each part of our implementation achieves this.

How It Works

Firstly, we will break down how our implementation works. Our implementation uses asymmetric encryption, using RSA with PKCS#1 v1.5 padding scheme for encryption. At the very beginning of the program, we generate 2 pairs of keys. A pair on the Python side, and a pair on the Java side. Then we do a key exchange so that

both sides have each other's public keys. These two pairs of keys are then used for both signing and encryption.

Before a message is sent through the protocol, the plaintext will first be encrypted using the sender's private key into ciphertext. This will make up one part of the message to be sent. For the other part, we hash the plaintext using the SHA-256 algorithm to obtain the hash. Then this hash is signed using the private key of the sender to obtain a signature. This signature is then concatenated to the ciphertext with a comma between them. These two parts make up the message and then the entire message will be encoded into Base64 before it is sent.

When received on the other side it is first decoded and split. After this, we decrypt the message to plaintext using the corresponding private key and verify the signature with the corresponding public key. By doing all this we ensure that the transfer of messages between the two is secure.

Encrypting it ensures that we satisfy the **Confidentiality** of the message, as only the two people who have the corresponding keys will be able to decrypt it and extract the content of the message.

Using signatures ensures that we satisfy the goals of **Integrity**, **Authenticity,** and **Accountability**. The **Integrity** and **Authenticity** of the message are maintained by verification of the signature. This means that if the message has been tampered with or if sent from another party the verification will fail. Only the sender and the receiver with the corresponding keys can verify the message successfully. This also satisfies **Accountability** as after the message is signed, one cannot deny signing it.

Advantages and Disadvantages

One positive of this implementation is that it satisfies **Accountability** due to the use of signatures. Asymmetric encryption also makes it easier for verification, authentication and supports **Non-repudiation**. And lastly, using multiple private keys makes the protocol more secure compared to just using a single private key in symmetric encryption.

The cons of our implementation are that it is slower and less efficient, due to asymmetric encryption and signatures. It is also more complicated to implement compared to a simple symmetric encryption protocol and is very resource heavy. This means that it is not suitable for encrypting large amounts of data as it would use more storage space and computing power.

However, the advantages outweigh the disadvantages in our scenario for developing a secure protocol for the MCPI API. Thus, it became our choice of implementation for this project.